第2章

初めてでも使える Verilog HDL文法ガイド《改訂版》

Verilog HDL 2001仕様に合わせて改訂

小林 優

SystemVerilogの詳細な説明に入る前に，従来のVerilog HDLの文法についておさらいする．本稿は，本誌2002年5月号の別冊付録に収録された「初めてでも使えるHDL文法ガイド Verilog HDL編」をVerilog HDL 2001仕様に合わせて改訂したものである．Verilog HDL 2001では，それまで文法的にあいまいとされてきた部分などが修正されている．（編集部）

Verilog HDLは，1995年にIEEE 1364として標準化されましたが，その後さまざまな修正と拡張が行われ，2001年にIEEE 1364-2001として新しい標準になりました．既存の機能もそのまま使えるように，仕様追加の形で改訂されています．

そこで，新たにHDL設計を始める方でもわかるように，Verilog HDL 2001の文法を「記述スタイル編」と「文法ガイド編」に分けて解説します．なお，記述スタイル編についてですが，新旧両方のスタイルが可能な場合には併記せず，新スタイルのみを紹介しています．文法ガイド編では両方を併記しています．

記述スタイル編

1 モジュール構造

● 基本構造

回路を記述する基本構造がモジュールです（**図1**）．モジュールは予約語のmoduleとendmoduleで囲まれ，回路表現からテストベンチ（検証用の記述）まで，すべてこの中で記述されます．moduleに続きモジュール名，ポート・リストを記述します．モジュール名やポート名などの識別子には英数字と_（アンダ・スコア）が使え，大文字小文字を区別します．

```
module モジュール名(
ポート・リスト
);

    変数宣言
    ネット宣言
    パラメータ宣言

    モジュール構成要素
        assign
        function
        always
        generate
        下位モジュール接続など

endmodule
```

図1 モジュール構造
回路記述もテストベンチも，すべてこのモジュール構造で記述する．さらに1モジュールは1ファイルで記述し，ファイル名とモジュール名を一致させておくとよい．

ポート・リストでは，入力/出力の方向，データ型，ビット幅，端子名を定義します（**リスト1(a)**）．双方向ポートはinoutを用います．ビット幅の表現は[0:31]や[32:1]も可能ですが，[MSB:LSB]として扱います．ポートの宣言をかっこの外で定義するスタイルもあります（2.6節の「generateによる回路の繰り返し」のところで説明）．

● 宣言部でネット信号や変数などを宣言

モジュールの最初に，内部で使用する信号の宣言を行います．

ネット宣言（**リスト1(b)**）では，回路記述で使用するネットをあらかじめ宣言します．ネット宣言では信号強度や

リスト1 各種宣言例

```
module MODULE_NAME (
input  wire   CLK, RST,            // 入力
input  wire   [15:0] BUS1, BUS2,   // 入力バス信号
output reg    BUSY,                // 出力
inout  wire   [31:0] DBUS          // 双方向バス信号
);
```

(a) ポート・リスト

```
wire   ENBL;
wire   [15:0] BUS;           // バス信号
wire   signed [31:0] DOUT;  // 符号付き32ビット
wire   #3000 TRIG;           // 遅延付加ネット
wire   (strong0, pull1) #100 OC_OUT = TRIG;
                               // オープン・コレクタ
```

(b) ネット宣言

```
reg  FF1, FF2;              // フリップフロップ
reg  [3:0] CNT4;            // 4ビット・カウンタ
reg  [7:0] MEM [0:1023];   // 1Kバイト・メモリ
```

(c) 変数宣言

```
parameter  STEP=1000;          // 1クロック周期
parameter  WAIT=2'b00, INIT=2'b01, ACTION=2'b10;
                                // ステート名
parameter  MEMSIZE=1024;       // メモリ・サイズ
reg   [7:0] MEM [0:MEMSIZE-1];  // 1Kバイト・メモリ
```

(d) パラメータ宣言

遅延を付加することもできます．ネット型にはwireのほかに**表3**(本稿の文法ガイド編，p.51)に示すようなものがあります．

多ビットの信号は，デフォルトで符号なしとして扱います．符号付きの信号にする場合には，予約語のsignedを付加します．これにより，演算時に符号拡張などが自動で行われます．

変数宣言(**リスト1(c)**)では，フリップフロップやラッチなどの値を保持する信号を定義します．メモリは配列として定義します．メモリをビット単位でアクセス(代入や参照)する場合は，mem[0][5]のようにします．これは，0番地のビット5です．変数には**表2**(文法ガイド編，p.51)に示すタイプがあります．integerはテストベンチの中でよく用いられます．

パラメータは定数の代わりの識別子として用います(**リスト1(d)**)．例えば1クロックの周期やメモリ・サイズなどです．値を変えてシミュレーションし直すときに重宝です．また，ステート・マシンの状態名に使うことで記述の可読性が向上します．

● コメントは2種，フリー・フォーマット

コメントは，

- /*～*/ … 複数行コメント
- //で始まり行末まで … 1行コメント

があります．フリー・フォーマットなので，記述を見やすくするためにスペースやタブ，改行を自由に挿入できます．

2 RTL記述

RTLとは，意訳すれば「詳細なブロック図レベル」です．ANDゲートやORゲートを記述したりフリップフロップを並べるのではなく，セレクタやカウンタのレベルで記述します．これがRTLです．

RTLの記述には，以下の五つの記述スタイルがあります．

- assignによる組み合わせ回路
- functionによる組み合わせ回路
- alwaysによる組み合わせ回路
- alwaysによる順序回路
- 下位モジュール接続

2.1 assignによる組み合わせ回路

論理式1行で記述できる組み合わせ回路は，assignで記述できます．**図2**に例を示します．信号が1ビットか多ビットかは宣言で決まります．したがってセレクタや加算回路などの記述はビット数に影響されません．

assignにより，AND，ORなどのゲート回路から，セレクタ，加算回路なども記述できます．関係演算子(==や<=など)は真のとき‘1’，偽のとき‘0’となるので，**図2**のけた上がり信号(CARRY)のようなこともできます．また，ネット型の信号宣言と，この信号に対する代入を一度に記述することができます(**リスト2**)．

各代入文は同時に実行されます．記述が前後しても動作に違いはありません．「代入」というより「接続」と考えたほうがわかりやすいでしょう．

```
assign  NA    = ~( IN1 & IN2 );          // 2入力NAND
assign  DOUT  = (SEL==1'b1) ? D1: D0;    // セレクタ
assign  CARRY = (CNT10==4'h9);           // けた上がり信号
assign  SUM   = A + B;                   // 加算回路
```

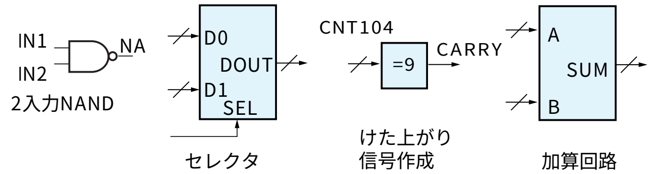


図2　assignによる組み合わせ回路
assignだけでどこまで回路が書けるか挑戦しているような，assignだらけの記述を見たことがある．assignの多用は可読性が悪く，保守性が悪くなるのでご注意を．if～elseやcaseを使うとすっきり書けることが多い．

リスト2　宣言と同時に回路を記述

```
wire [15:0] A, B;
wire [15:0] SUM = A +
```

リスト3　3 to 8デコーダ

```
module DEC3TO8_FUNC (
input  wire [2:0] DIN,
output wire [7:0] DOUT
);

function [7:0] DEC (
input [2:0] DECIN
);
begin
    case ( DECIN )
        3'h0:    DEC = 8'b0000_0001;
        3'h1:    DEC = 8'b0000_0010;
        3'h2:    DEC = 8'b0000_0100;
        3'h3:    DEC = 8'b0000_1000;
        3'h4:    DEC = 8'b0001_0000;
        3'h5:    DEC = 8'b0010_0000;
        3'h6:    DEC = 8'b0100_0000;
        3'h7:    DEC = 8'b1000_0000;
        default: DEC = 8'bx;
    endcase
end
endfunction

assign DOUT = DEC( DIN );

endmodule
```

（a）functionによるデコーダ

```
module DEC3TO8_ALWAYS (
input  wire [2:0] DIN,
output reg  [7:0] DOUT
);

always @*
begin
    case ( DIN )
        3'h0:    DOUT = 8'b0000_0001;
        3'h1:    DOUT = 8'b0000_0010;
        3'h2:    DOUT = 8'b0000_0100;
        3'h3:    DOUT = 8'b0000_1000;
        3'h4:    DOUT = 8'b0001_0000;
        3'h5:    DOUT = 8'b0010_0000;
        3'h6:    DOUT = 8'b0100_0000;
        3'h7:    DOUT = 8'b1000_0000;
        default: DOUT = 8'bx;
    endcase
end

endmodule
```

（b）alwaysによるデコーダ

2.2　functionによる組み合わせ回路

論理式1行で記述できないような組み合わせ回路，例えばデコーダやROMなどはfunctionを用いて記述します．ファンクション内では，if～elseや，caseなどの制御構造を使用できるので，複雑な組み合わせ回路を簡潔に記述できます．**リスト3(a)**にfunctionによる3 to 8デコーダの記述を示します．3ビット入力DINの値をデコードし，8ビット出力DOUTの中の1ビットだけが'1'となる回路です．

ファンクションは，予約語functionの後に戻り値のビット幅，ファンクション名，そしてファンクションの入力宣言と続きます．このファンクションの本体は，case文が一つだけです．入力DINに対して8通りに分岐し，一致した文を実行します．ファンクションの戻り値は，ファンクション名に直接代入することで行われます．case文，ファンクションはそれぞれendcase，endfunctionで終わります．

複数の文をまとめて一つの文として扱うときはbegin～endを使います．**リスト3**の例ではファンクション内はcase文しかありませんが，念のため付けておきました．文の末尾にはかならず;（セミコロン）を付けます．しかし，end～の予約語で終わっている場合は不要です．

ファンクションは定義側なので，呼び出し側が必要です．assignでファンクションDECを呼び出し，DOUTに代入してデコーダの記述は完了です．

2.3　alwaysによる組み合わせ回路

組み合わせ回路をalwaysで記述することも可能です．先ほどのデコード回路を，alwaysを用いて記述してみました（**リスト3(b)**）．

alwaysによる組み合わせ回路の要点は，

- 出力をreg宣言する
- alwaysの@以降（イベント式ともいう）にすべての入力を記述する

リスト4　ラッチの記述

```
module LATCH  (
input  wire G, D,
output reg  Q
) ;

always @* begin
    if  ( G==1'b1 )
        Q = D;
end

endmodule
```

リスト5　リセット付きDフリップフロップの記述

```
module DFF_SYNC_RST (
input  wire CK, RB, D,
output reg  Q
);

always @( posedge CK ) begin
    if ( RB==1'b0 )
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```

(a) 同期リセット

```
module DFF_ASYNC_RST (
input  wire CK, RB, D,
output reg  Q
);

always @( posedge CK, negedge RB ) begin
    if ( RB==1'b0 )
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```

(b) 非同期リセット

●alwaysの中に動作(出力に対する代入文)を記述する

となります.

alwaysは「無限ループ」であり,@は「待ち」を作ります.@以降に記述された信号に変化があるときに,「待ち」が解除され,always内部が実行されます.つまり,入力が変化すれば,出力に対する代入文が実行されることになります.結局,組み合わせ回路になります.

リスト3(b)では@*と記述していますが,この * は,「入力信号のいずれか」を意味します.つまり,

@(入力信号, 入力信号, 入力信号, ‥)

とするところを省略して記述しています.多数の入力がある場合,簡潔に記述できます.

2.4　alwaysによる順序回路

順序回路とは,値を保持するフリップフロップやラッチを含む回路です.alwaysを使って記述します.組み合わせ回路とは記述スタイルが若干異なります.

● ラッチ(レベル・センシティブ)

ラッチは**リスト4**のように記述します.@*が使われているので,入力GとDのいずれかが変化したとき,begin～end内を実行します.一見組み合わせ回路の記述と同じですが,値を保持するラッチになります.動作を説明します.

内部のif文により,

●G＝1　…　DからQへデータが筒抜け

●G＝0　…　Dが変化してもQは変化しない

となり,値を保持します.

組み合わせ回路になるかラッチになるかは,じつは紙一重です.組み合わせ回路を記述したつもりでも,値を保持するような記述の場合,論理合成ツールは忠実にラッチを生成してしまいます.

● フリップフロップ(エッジ・センシティブ)

Dフリップフロップの記述例を**リスト5**に示します.クロックCKの立ち上がりでデータを取り込み,リセットRBが'0'のときリセットします.**リスト5(a)**は同期リセット付き,**リスト5(b)**は非同期リセット付きです.

同期リセット付きでは,出力が変化する条件はCKの立ち上がり(positive edge)だけなので,always文のイベント式は(posedge CK)だけです.そしてRBが'0'ならリセットし,'0'でなければD入力の値をQ出力に伝えます.

一方,非同期リセット付きでは,RBの立ち下がり(negative edge)もイベント式に含まれ,CKとは非同期にリセットします.結局,同期か非同期の違いは,イベント式に含むか含まないかの違いだけです.

● 代入記号は2種類ある

値を保持する変数には,＝と<=の2種類の代入記号を使うことができます.＝は順次処理向き,<=は同時処理に向いた代入記号です.一般に,順序回路の記述では<=,組み合わせ回路の記述やテストベンチでは＝を使います.

● カウンタ,シフト・レジスタ

フリップフロップの応用として,カウンタとシフト・レジスタの例を示します.**リスト6(a)**はロード優先のイネーブル付きカウンタです.ロード信号LDが'1'ならパラレル・ロードし,イネーブル信号ENが'1'ならカウント・アップします.**リスト6(b)**はイネーブル付きのシリアル-パラレル変換シフト・レジスタです.連接演算を使ってシフ

リスト6 カウンタとシフト・レジスタ

```
module COUNTER4 (
input  wire       CK, RB, LD, EN,
input  wire [3:0] D,
output reg  [3:0] Q
);

always @( posedge CK, negedge RB ) begin
    if ( RB==1'b0 )
        Q <= 4'h0;
    else if ( LD==1'b1 )
        Q <= D;
    else if ( EN==1'b1 )
        Q <= Q + 4'h1;
end

endmodule
```

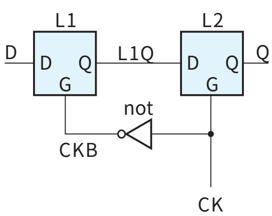
(**a**) ロード，イネーブル付きカウンタ

```
module SHIFT4 (
input  wire       CK, RB, SI, EN,
output reg  [3:0] Q
);

always @( posedge CK, negedge RB ) begin
    if ( RB==1'b0 )
        Q <= 4'h0;
    else if ( EN==1'b1 )
        Q <= {Q[2:0], SI};
end

endmodule
```

(**b**) イネーブル付きシフト・レジスタ

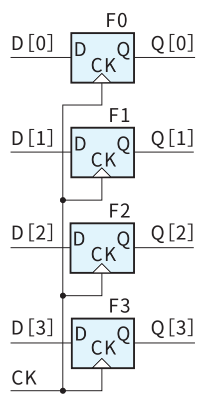


(**a**) ラッチ接続による
フリップフロップ

```
module FF(
input  CK,D,
output Q);
  wire  L1Q, CKB
  assign CKB = ~CK;
  LATCH  L1(CKB, D, L1Q);
  LATCH  L2(CK, L1Q, Q);
endmodule
```

(**b**) ラッチ接続によるフリップフロップの
Verilog HDL記述

図3 フリップフロップ回路
ここで示したフリップフロップは，モジュール接続の例として出した．実際のフリップフロップは，リスト7に示したようにalwaysを使って記述する．



(**a**) 4ビット・
フリップフロップ

```
module FF4(
input        CK,
input  [3:0] D,
output [3:0] Q );
  FF F0( .D(D[0]), .Q(Q[0]),
                   .CK(CK) );
  FF F1( .D(D[1]), .Q(Q[1]),
                   .CK(CK) );
  FF F2( .D(D[2]), .Q(Q[2]),
                   .CK(CK) );
  FF F3( .D(D[3]), .Q(Q[3]),
                   .CK(CK) );
endmodule
```

(**b**) 4ビット・フリップフロップの
Verilog HDL記述

図4 4ビット・フリップフロップ
名まえによる接続は順番に依存しないので，誤りの入る余地が少ない．確実な検証を目指すのならこれを使うべきだが，記述の手間がかかるのが難である．Perlスクリプトなどを使ってポート・リストを自動生成する簡易ツールを作った例もあるようだ．

ト動作を実現しています．

2.5 下位モジュール接続

別途定義したモジュールを，上位階層で接続することもできます．機能的にまとまった単位でモジュール化し，これらを接続して一つの階層を作り，さらにこれらを上位階層で接続して…というような階層設計が可能です．

図3では，**リスト4**のラッチを二つ接続してフリップフロップを作成しています．内部で使用する二つの信号L1QとCKBは値を保持しないので，ネット型(wire)で宣言しています．

下位モジュールの接続は次のように行います．接続するモジュール名に続き，インスタンス名(この場合はL1，L2)を記述します．さらにかっこ内に接続する信号をコンマで区切って並べます．記述順にモジュール定義側と接続されます．順番をまちがえると，正しく接続されません．

図4はこのフリップフロップを4個接続して4ビット・フリップフロップを構成したものです．ここでは入出力に名まえによる接続を行っています．

```
.定義側ポート名(接続信号名)
```

とすることで順番が任意になります．モジュール接続時にポート数が数十になることも珍しくないので，順番が任意であることは有用です．ただし，順番による接続と名まえによる接続は混在できません．

2.6 そのほかのRTL記述

● generateによる回路の切り換え

図3のフリップフロップの記述は確かに動作しますが，現実的ではありません．フリップフロップには適した記述があり，通常alwaysを使います．これら2種類の記述をgenerateとif文を用いて切り換えてみました(**リスト7**)．

generate～endgenerate内にif～elseの構造で2

リスト7　generateとif文

```
module FF(
input  CK, D,
output Q );

parameter ff_type = 0;

generate
if (ff_type==0) begin
    /* ラッチ接続によるFF */
    wire L1Q, CKB;
    assign CKB = ~CK;
    LATCH L1 ( CKB, D, L1Q );
    LATCH L2 ( CK,  L1Q, Q );
end
else if (ff_type==1)  begin
    /* alwaysによるFF */
    reg  Q_reg;
    always @( posedge CK ) begin
        Q_reg <= D;
    end
    assign Q = Q_reg;
end
endgenerate

endmodule
```

リスト8　generateとfor文

```
module FF4( CK, D, Q );
parameter width = 4;
input              CK;
input  [width-1:0] D;
output [width-1:0] Q;

generate
    genvar i;
    for (i=0; i<width; i=i+1 ) begin: ff_loop
        /* FFの接続 */
        FF F0 ( .D(D[i]),  .Q(Q[i]), .CK(CK) );
    end
endgenerate

endmodule
```

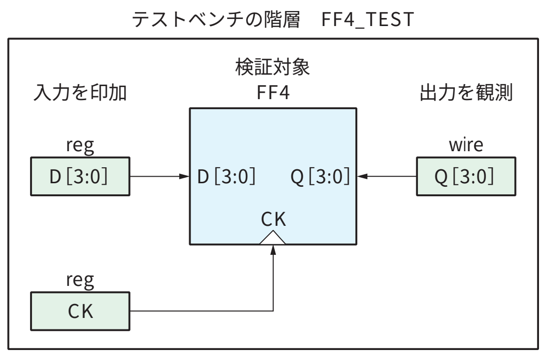


図5　シミュレーション時のモジュール構造
検証対象の一つ上の階層でテストベンチを作成する．ここで入力を与えて出力を観測するしくみを作る．

種類のフリップフロップを記述し，パラメータ宣言した定数ff_typeによって切り換えています．generate内には回路記述で使う大半の構文を記述できます．ポートやモジュール内で宣言した信号にアクセスできるので，CKやD入力を参照し，出力Qに信号を与えています．

parameterは定数なので，このままではつねにラッチ接続のフリップフロップになってしまいますが，上位階層でこのモジュールを接続するときに，

```
FF #(.ff_type(1)) (CK, D, Q);
```

とすることで，ff_typeに値を与えることができます．parameterはデフォルト値として扱われます．

● generateによる回路の繰り返し

generate内にはfor文を記述できるので，これを利用することで回路を繰り返し接続できます．**リスト8**は，**図4**をgenerateを使って書き換えたものです．generate内部にローカルな変数iをgenvarを使って宣言し，for文のループ変数にしています．これを接続する多ビット信号のビット選択に使っています．

この記述では，ビット幅をパラメータ宣言を使って定義しました．前の例と同じように，このモジュールを上位階層で接続するときに任意のビット幅にすることができます．パラメータ宣言は，ポート・リストのかっこ内に位置できないので，今までと異なったスタイルでポートを宣言しています．モジュール名に続き，かっこ内でポート名だけをリストし，モジュール内でポートの方向やビット幅を定義しています．

3 テストベンチの基本

回路記述はシミュレーションにより検証します．このためには，検証対象となる回路に信号を与え，状態を観測するための記述が必要です．これがテストベンチです．設計した回路（モジュール）のファイル内にテストベンチを含めることも可能です．しかし，後々論理合成を用いてFPGAやASICに実装するので，**図5**のようにテストベンチは階層に分けて記述するのが一般的です．逆に言えば，テストベンチは論理合成対象外なので，Verilog HDLの記述力を最大限に利用できます．

リスト9に**図4**の4ビット・フリップフロップをテストするためのテストベンチを示します．記述のための必要事項を順を追って説明します．

リスト9　4ビット・フリップフロップのテストベンチ

```
module FF4_TEST;
reg        CK;          // 入力にはreg宣言した信号を接続
reg  [3:0] D;
wire [3:0] Q;           // 出力にはwire宣言した信号を接続

parameter STEP = 1000;  // 1周期を1000ユニットに

always begin            // クロックの作成
    CK = 0; #(STEP/2);
    CK = 1; #(STEP/2);
end

FF4 FF4( CK, D, Q );    // 検証対象の接続

initial begin
            D = 4'h0;   // 入力の設定
    #STEP   D = 4'h5;
    #STEP   D = 4'ha;
    #STEP   D = 4'hf;
    #STEP   $stop;
end

endmodule
```

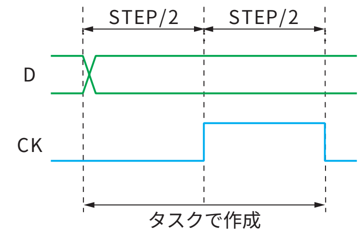


図6　入力波形
タスクは効果的な検証には欠かせない手段．バス動作を記述したり，入出力のシーケンスを記述したりと，用途は無数にある．

1）ポートのないモジュール

この階層は最上位階層に位置するので，外部とやり取りするポートは不要です．ポートがありませんから，モジュール名の後はセミコロンで終わります．

2）入力を用意する

検証対象回路に外部から信号を与えるため，変数宣言で信号を定義しておきます．**リスト9**では，FF4に与えるクロックとD入力を変数宣言しています．一方，出力はwireを使ってネット宣言しておきます．

また，1クロックの周期もパラメータ宣言を用いて定義しておきましょう．この場合，1ステップ1,000ユニットです．ユニットとは，シミュレーション時の時間の最小単位です．ユニットと実時間の対応は，コンパイラ指示子の`timescaleを使います(文法ガイド編を参照)．

3）クロックを作成する

シミュレーション中つねにクロックを発生させるためには，alwaysを用いて記述します．#は遅延を意味します．

- CKを‘0’にして半周期遅延
- CKを‘1’にして半周期遅延

これを繰り返すことで，デューティ50％のクロックCKを作成できます．遅延量をくふうすれば，デューティを変えたり，多相クロックを作ることも可能です．

4）検証対象を接続する

検証対象に入出力の信号を接続します．あらかじめ宣言しておいた変数やネットを接続します．複数のモジュールを接続することにより，複数ブロックの検証も可能です．

5）入力を与える

initialを用いて入力を順次与えます．initialは，alwaysと対称的に，シミュレーション開始後に1回だけ実行します．プログラム言語に近い動作です．ただし，alwaysやinitialはいくつでも記述できますから，並列動作も可能です．

initialの中でD入力を変化させるたびに1周期の遅延を与えています．遅延を与えることでシミュレーションが進みます．最後にシステム・タスク$stopで停止します．

システム・タスクは$で始まり，**表6～表8**(文法ガイド編，p.56)に示すようなものが用意されています．ファイル・アクセスが豊富にあるので，シミュレーション結果をすべてファイルに書き出すことや，よそで作成したデータとの自動比較も可能です．

4 テストベンチ向き構文

設計した回路を効率的に検証するために，Verilog HDLの記述能力を最大限利用することができます．テストベンチは，ある意味で「プログラミング」です．プログラミングに必要な構文が，Verilog HDLには多数用意されています．

● サブルーチンに相当する「タスク」

テスト入力の作成や，期待値との照合にサブルーチンに相当する概念のタスクを用いると，検証効率を向上できます．

タスクはサブルーチンの一種ですから，タスクの定義やタスクの呼び出しがあります．また，呼び出し側とのデータの受け渡しを行う「引き数」もあります．さらに，タスクの中で別のタスクを呼び出す「ネスティング」も可能です．

リスト9のテストベンチを，タスクを用いて記述し直してみました．**図6**のように，クロック1周期分の期間でD

入力とクロックCKを制御し，フリップフロップへ任意の値を書き込むタスクを記述してみました（リスト10）．

タスクは，taskに続いてタスク名と引き数の宣言を記述します．この例では引き数が入力（input）ですが，出力の引き数（output）も可能です．

タスクをサブルーチンとすれば，モジュール内で宣言した信号はグローバル信号です．したがって，モジュール内で宣言した信号をタスク内で参照できます．ここでは，テスト対象に接続しているDやCKに対し，タスク内から値を設定しています．D入力に対して，引き数のdataを代入しています．

タスク内では遅延も記述できます．ここではSTEP/2単位で信号を変化させ，図6の入力を実現しています．

タスクは，システム・タスクと同じようにinitial内で呼び出します．リスト10に示すようにタスクの呼び出しを記述します．ここでは4ビットの定数を連続的に4ビット・フリップフロップに書き込んでいます．

このタスクでは，シミュレーション開始直後からSTEP/2ユニット後にCKの値が確定します．それ以前は不定値です．これを防ぐため，CKの宣言時に'0'を初期値として与えています．変数宣言時の初期値設定はとても便利ですが，回路記述内で使うことは禁止です．多くの場合，論理合成ツールで無視されます．レジスタ類の初期化はリセット機能としてきちんと記述する必要があります．

● ループ構文など

ループを作る構文には，for文，while文，repeat文，forever文などがあります．また，強制代入のforce文やforce文の解除のrelease文など，さまざまな構文があります．詳細については，文法ガイド編を参照してください．

● システム・タスク

シミュレーション結果を表示したりファイルに出力するため，さまざまなシステム・タスクが用意されています．とくに，ファイルの入出力については，C言語と同様の名称と機能があります．

● コンパイラ指示子

module～endmoduleの外部に記述でき，記述のコンパイル時に作用するさまざまな構文があります．バック・クォート（`）で始まるこれらの構文を「コンパイラ指示子」と呼びます．

\`define	テキスト・マクロ
\`timescale	シミュレーション時間の単位づけ
\`include	ファイル読み込み
\`ifdef，\`else，\`endif，\`elsif	条件コンパイル

などがあります．こちらも文法ガイド編で紹介しています．

リスト10　タスクを用いたテストベンチ

```
module FF4_TEST;
reg        CK = 0; /* 初期値設定 */
reg  [3:0] D;
wire [3:0] Q;

parameter STEP = 1000;

FF4 FF4( CK, D, Q );

/* タスクの定義 */
task FF4_write (
input [3:0] data );
begin
                D = data;
    #(STEP/2)   CK = 1;
    #(STEP/2)   CK = 0;
end
endtask

/* タスクの呼び出し */
initial begin
    FF4_write( 4'h0 );
    FF4_write( 4'h5 );
    FF4_write( 4'ha );
    FF4_write( 4'hf );
    #STEP $stop;
end

endmodule
```

こばやし・まさる
(株)エッチ・ディー・ラボ
koba@hdlab.co.jp

<筆者プロフィール>
小林 優．1996年，コンサルティング会社 エッチ・ディー・ラボの設立に参加．以後，各種セミナの企画，開発，講師などに従事．最近は組み込みにも手を出し，ソフトウェア技術者にハードウェアを教えることもしばしば．モノ作りが自分の原点と悟り，真空管アンプやらDAC基板を製作し自己満足している．

文法ガイド編

回路記述やテストベンチでよく用いるものについて，文法要約を示します．簡略化して表現したものもあります．また，省略できる項目には [ ] を付けました．

1 基本項目

1.1 識別子

● 通常の識別子

- 英字またはアンダ・スコア(_)で始まる文字列．
- 文字列中には，英字，数字，アンダ・スコア(_)，ドル記号($)を含むことができる．
- 大文字と小文字を区別する．

《記述例》

正しい識別子　cnt4，_reset，TEN$，INPUT
（大文字なので予約語の input とは区別する）
誤った識別子　74LS00(先頭が数字)，$test(先頭が $)，xor(予約語)

● エスケープされた識別子

- バック・スラッシュ(\)で始まる文字列（日本語環境では¥で始まる文字列）．
- 任意の印字可能な ASCII 文字を含むことができる．
- ホワイト・スペース（スペース，タブ，改行）が識別子の区切りとなる．

《記述例》

¥cnt4_reg[3] ，¥74LS04 ，¥/*Hello,HDL-World*/
（それぞれの末尾はスペース）

1.2 コメント，フォーマット

● コメント

- // で始まり，行末までの1行
- /* ～ */ で囲まれた複数行

《記述例》

```
// 入力はreg宣言
/* データ処理部
  by Dr. Kobanovski */
```

● ソース・フォーマット

ソース・フォーマットはフリー・フォーマットであり，空白，タブ，改行など自由に挿入できる．

1.3 論理値

- 0，1，x，z の4値．
- x(または大文字の X)：不定値
- z(または大文字の Z)：ハイ・インピーダンス
数値表現の中での?は z と同義．

1.4 数値表現

size'base value の形で表現する．

size ：定数のビット幅を示す10進数値．

'base ：基数を示す文字．d，h，o，b があり，それぞれ decimal(10進)，hexadecimal(16進)，octal(8進)，binary(2進)．
符号付き数値に場合には，それぞれ sd，sh，so，sb を使う．
以上は大文字でも可．

value ：定数値．各基数で許される値のほかに，x，z が記述できる．区切りのために _(アンダ・スコア)を使用できる．

ビット幅を省略すると32ビット幅，さらに基数も省略すると32ビット幅の10進数となる．

表1　数値表現例

数値定数	ビット幅	基数(進数)	2進数表現
10	32	10	000...000001010
1'b1	1	2	1
8'haa	8	16	10101010
4'bz	4	2	zzzz
8'o377	8	8	11111111
8'b0000_11xx	8	2	000011xx
'hff	32	16	000...011111111
4'd5	4	10	0101
8'sha	(符号付き) 8	16	11111010
'shfffe	(符号付き) 32	16	111...111111110
6'shf0	(符号付き) 6	16	110000

2 モジュール構造

```
module <モジュール名> (<ポート宣言>);
  <パラメータ宣言>
  <変数型宣言>
  <イベント宣言>
  <ネット型宣言>
  <プリミティブ・ゲート接続>
  <下位モジュール接続>
  <generateブロック>
  <alwaysブロック>
  <initialブロック>
  <function定義>
  <task定義>
  <継続的代入>
endmodule
```

注：モジュール構成要素(モジュール内の各要素)の記述順は任意．ただし，信号名やパラメータなどの識別子は，使用する前に宣言が必要．

3 モジュール構成要素

● ポート宣言

```
┌input  [<ネット型>]            [signed] [<レンジ>]
│                              <信号名>, <信号名>, …
├output [<ネット型>またはreg] [signed] [<レンジ>]
│                              <信号名>, <信号名>, …
└inout  [<ネット型>]            [signed] [<レンジ>]
                               <信号名>, <信号名>, …
```

● レンジ

```
[<定数式>:<定数式>]
```

注：[MSB:LSB]となる．レンジを省略すると1ビット．
　ここの[]は省略可能を意味するのではなく，[]そのものを意味する．

《記述例》

```
module BLOCK (
input  wire CK, RST, LD,
output reg  [3:0]  Q,
inout  wire [15:0] DBUS );
```

注：ポート宣言をかっこの外で行うスタイルもある．
　module <モジュール名> (ポート名, ポート名, …);
　<ポート宣言>;

《記述例》

```
module BLOCK ( CK, RST, LD, Q, DBUS );
input  CK, RST, LD;
output [3:0]  Q;
inout  [15:0] DBUS;
reg    [3:0]  Q;
```

● パラメータ宣言

```
┌parameter [signed] [<レンジ>] <パラメータ名>
│     = <定数式>, <パラメータ名> = <定数式>, … ;
└parameter integer             <パラメータ名>
      = <定数式>, <パラメータ名> = <定数式>, … ;
```

《記述例》

```
parameter STEP=1000;
parameter [1:0] HALT=2'b00, INIT=2'b01,
                ADRINC=2'b10, RAMWRT=2'b11;
parameter integer MINUS = -1234;
```

● 変数宣言

```
<変数型> [signed] [<レンジ>] <変数名リスト>;
```

● 変数型

表2　変数型

変数型名	機　能
reg	任意ビット（未指定時は符号なし）
integer	符号付き32ビット
time	符号なし64ビット
real	実数
realtime	実数表記での時間

● 変数名リスト

```
<信号名> [<次元> <次元> … ],
          <信号名> [<次元> <次元> … ], …
```

● 次元

```
[<定数式>:<定数式>]
```

《記述例》

```
reg FLAG, Q;     // 1ビット変数
reg [3:0] CNT;   // 4ビット変数
reg [7:0] ARRAY [0:255][0:15];
              // 8ビットの要素が256×16個ある2次元配列
```

● イベント宣言

```
event <イベント名リスト>;
```

● イベント名リスト

```
<イベント名> [<次元> <次元> … ],
          <イベント名> [<次元> <次元> … ], …
```

注：イベントは値を持たない信号．したがって，代入ではなく「起動」できる．イベント信号の起動には -> を使い，参照は @(〜) のイベント式の中で行う．

《記述例》

```
event EVENT_SIG1, EVENT_SIG2;
```

● ネット宣言

```
<ネット型> [signed] [<レンジ>] [<遅延>]
                              <ネット名リスト>;
```

● ネット型

表3　ネット型

ネット型名		機　能
wire	tri	通常のネット，wireとtriは同義
wor	trior	ワイヤードORネット，worとtriorは同義
wand	triand	ワイヤードANDネット，wandとtriandは同義
tri0	tri1	プルダウン，プルアップされたネット
supply0	supply1	電源ネット
trireg		電荷蓄積ネット

2

● ネット名リスト

```
< 信号名 > [< 次元 > < 次元 > … ],
                < 信号名 > [< 次元 > < 次元 > … ], …
```

● 遅延

```
┌#< 定数式 >
├#( <min_typ_max 定数式 > )
├#( <min_typ_max 定数式 >, <min_typ_max 定数式 > )
│  注：プリミティブ・ゲート接続用.（立ち上がり，立ち下がり）の
│     遅延を設定.
└#( <min_typ_max 定数式 >, <min_typ_max 定数式 >,
                           <min_typ_max 定数式 > )
 注：プリミティブ・ゲート（3ステート・タイプ）接続用.
    （立ち上がり，立ち下がり，ターン・オフ）の遅延を設定.
```

● min_typ_max 定数式

```
< 定数式 >:< 定数式 >:< 定数式 >
```

注：min:typ:max の遅延量を設定．どの値を使用するかは，シミュレーション時に設定する．

《記述例》

```
wire       X, Y, Z;    // 1ビットのネット
wire [3:0] Q;          // 3ビットのネット
wire [7:0] #(10:15:20) DBUS;
                       //遅延付きの8ビット・バス
```

● プリミティブ・ゲート接続

```
< ゲート・タイプ > [< 信号強度 >] [< 遅延 >]
        [< ゲート名 >] (< 信号名 >, < 信号名 >, … );
```

● ゲート・タイプ

表4 ゲート・タイプ

ゲート	3ステート	スイッチ		プルアップ，プルダウン
and	bufif0	nmos	tran	pullup
nand	bufif1	pmos	tranif0	pulldown
nor	notif0	cmos	tranif1	
or	notif1	rnmos	rtran	
xor		rpmos	rtranif0	
xnor		rcmos	rtranif1	
buf				
not				

● 信号強度

```
┌( < 強度 0>, < 強度 1> )
└( < 強度 1>, < 強度 0> )
```

表5 信号強度

信号強度		強度レベル
supply0	supply1	7
strong0	strong1	6
pull0	pull1	5
large0	large1	4
weak0	weak1	3
medium0	medium1	2
small0	small1	1
highz0	highz1	0

《記述例》

```
nand (weak0, weak1) #(5:10:15)
              N1 (OUT2, IN0, IN1); // NANDゲート
```

● 下位モジュール接続

```
< モジュール名 > [< パラメータ割り当て >] < インスタンス名 >
                                    (< ポート・リスト >) ;
```

注：モジュール名と，インスタンス名を同じにしても文法上OK.

● パラメータ割り当て

```
┌#( < 式 >, < 式 >, … )
└#( .< パラメータ名 >(< 式 >),
                  .< パラメータ名 >(< 式 >), … )
```

注：モジュール内の parameter 宣言した定数に対し，呼び出し時に上書き設定できる．HDL ライブラリ作成に有用．

● ポート・リスト

```
┌< 信号名 >, < 信号名 >, < 信号名 >, …
└.< 定義側ポート名 > ( < 信号名 > ),
              .< 定義側ポート名 > ( < 信号名 > ), …
```

注：ポート・リストの < 信号名 > は，正確には < 式 >．したがって定数でも論理式でも記述できる．

《記述例》

```
ADDER ADDER( A, B, Q );
FF #(4) FF4( .CK(CLK), .D(DIN), .Q(DOUT) );
SUBCIRCUIT S1( ADDR[15:12], FLAG & BUSY, 1'b1 );
```

● generate ブロック

```
generate
  <genvar 宣言 >
  < パラメータ宣言 >
  < 変数型宣言 >
  < イベント宣言 >
  < ネット型宣言 >
  < プリミティブ・ゲート接続 >
  < 下位モジュール接続 >
  <if 文 >
  <case 文 >
  <for 文 >
  begin 〜 end
```

```
  <always ブロック>
  <initial ブロック>
  <function 定義>
  <task 定義>
  <継続的代入>
endgenerate
```

注：ここでの for 文には，begin: <ブロック名> ～ end を付加する．

● genvar 宣言

```
genvar <変数名>, <変数名>, …;
```

● always ブロック

```
always <ステートメント>
```

《記述例》

```
always @( posedge CK )   // 順序回路
    Q <= D;
always @( A, B, C )      // 組み合わせ回路
    Y <= A & B & C;
always begin             // クロックの記述
    CK = 1; #(STEP/2);
    CK = 0; #(STEP/2);
end
```

● initial ブロック

```
initial <ステートメント>
```

《記述例》

```
initial begin // テスト入力
              CK  = 0;
    #STEP     RST = 1;
    #STEP     RST = 0;
    # (STEP*20)  $finish;
end
```

● function 定義

```
function [automatic] [signed] [<レンジまたは型>]
 <ファンクション名>(<ファンクション・ポート宣言>);
  <タスク・ファンクション内宣言>
  <ステートメント>
endfunction
```

注：automatic を付加すると再帰呼び出しが可能となる．

● レンジまたは型

```
├<レンジ>
└<タスク・ポート型>
```

● タスク・ポート型

```
├integer
├real
├realtime
└time
```

● ファンクション・ポート宣言

```
├input [reg] [signed] [<レンジ>]
│                     <ポート名>, <ポート名>, …
└input [<タスク・ポート型>]
                      <ポート名>, <ポート名>, …
```

● タスク・ファンクション内宣言

```
├<パラメータ宣言>
├<変数型宣言>
└<イベント宣言>
```

《記述例》

```
// 2 to 4 デコーダ
function [3:0] dec (input [1:0] din);
    case ( din )
        0: dec = 4'b0001;
        1: dec = 4'b0010;
        2: dec = 4'b0100;
        3: dec = 4'b1000;
        default:
           dec = 4'bx;
    endcase
 endfunction
```

● task 定義

```
task [automatic] <タスク名> (<タスク・ポート宣言>);
 <タスク・ファンクション内宣言>
 <ステートメント>
endtask
```

● タスク・ポート宣言

```
├input  [reg] [signed] [<レンジ>]
│                     <ポート名>, <ポート名>, …
├input  [<タスク・ポート型>]
│                     <ポート名>, <ポート名>, …
├output [reg] [signed] [<レンジ>]
│                     <ポート名>, <ポート名>, …
├output [<タスク・ポート型>]
│                     <ポート名>, <ポート名>, …
├inout  [reg] [signed] [<レンジ>]
│                     <ポート名>, <ポート名>, …
└inout  [<タスク・ポート型>]
```

```
              < ポート名 >, < ポート名 >, …
```

注：task内では 遅延(#～, @～)を記述できるが, function内では不可.

《記述例》

```
task FF_write( input [3:0] data );
  begin
                  D  = data;
    #(STEP/2)  CK = 1;
    #(STEP/2)  CK = 0;
  end
endtask
```

● 継続的代入

```
┌assign [< 信号強度 >] [< 遅延 >] < ネット型信号名 >
│           = < 式 >, < ネット型信号名 > = < 式 >, … ;
└< ネット型 > [< 信号強度 >] [signed] [< レンジ >]
           [< 遅延 >] < ネット型信号名 > = < 式 >, … ;
```

《記述例》

```
assign # (5:10:15)  sum = a + b;
wire (pull1, pull0) [15:0] tribus = reg_A[15:0]
          + reg_B[15:0]; // 信号強度付き信号への代入
```

4 ステートメント

● タイミング・コントロール(各ステートメントの直前に記述できる)

```
├< イベント制御 >
├#< 定数式 >
├#( <min_typ_max 定数式 > )
└repeat ( < 式 > ) < イベント制御 >
```

注：この repeat は, repeat 文とは異なり, タイミング・コントロールの拡張とみなされる.

《記述例》

```
repeat ( 8 ) @( posedge CK ) TRIG = 1;
                // 8 クロック待って TRIG を 1 に
```

● イベント制御

```
├@< 信号名 >
├@( < イベント式 > )
├@*
└@(*)
```

● イベント式

```
├< 式 >
├posedge < 式 >
├negedge < 式 >
├< イベント式 > or < イベント式 >
└< イベント式 >, < イベント式 >
```

● ブロッキング代入文

```
< 変数左辺 > = [< タイミング・コントロール >] < 式 >;
```

● 変数左辺

```
├< 変数 >
├< 変数 > [< レンジ式 >]
├< 変数 > [< 式 >] [< 式 >] ... [< 式 >]
├< 変数 > [< 式 >] [< 式 >] ... [< 式 >] [< レンジ式 >]
└< 連接 >
```

注：配列の [] は, 宣言時に確保した次元の数だけ付加する.
ここの [] は省略可能を意味するのではなく, [] そのものを意味する.

● レンジ式

```
├< 式 >
├< 定数式 >:< 定数式 >
├< 式 >+:< 定数式 >
└< 式 >-:< 定数式 >
```

《記述例》

```
wire  [7:0]   BYTE;
wire  [31:0]  WORD;
reg   [1:0]   POS;
// POS=0 のとき WORD[7:0]を, 1 のとき WORD[15:8]を選択
assign BYTE = WORD[ POS*8 +: 8 ];
```

● 連接

```
├{ < 式 >, < 式 >, … }
└{ < 定数式 > { < 式 >, < 式 >, … } }
```

注：{} 内を定数式の値だけ繰り返す.

《記述例》

```
{ {16{BUS[15]}}, BUS }
          // 16 ビットから 32 ビットへの符号拡張
```

● ノン・ブロッキング代入文

```
< 変数左辺 > <= [< タイミング・コントロール >] < 式 >;
```

注：ブロッキング代入文は, 一つの代入処理が終了するまで次の文を実行しない.
ノン・ブロッキング代入文は各右辺の処理が終了してから代入処理が行われる.

● ブロック

```
├begin < ステートメント > end
├begin : < ブロック名 > < ブロック内宣言 >
│                        < ステートメント > end
```

```
┌fork  ＜ステートメント＞ join
└fork  ：＜ブロック名＞ ＜ブロック内宣言＞
                      ＜ステートメント＞ join
```

注：複数の文をまとめて一つの文として扱う．
　begin～end(順序処理ブロック)…　記述順に実行．
　fork～join(並列処理ブロック)…　並列に実行．

● ブロック内宣言

```
┌＜パラメータ宣言＞
├＜変数型宣言＞
└＜イベント宣言＞
```

● if 文

```
┌if （＜式＞）＜ステートメント＞
└if （＜式＞）＜ステートメント＞ else ＜ステートメント＞
```

《記述例》

```
if ( RST ) //カウンタの本体
    Q <= 4'b0;
else
    Q <= Q + 4'h1;
```

● case 文(casex，casez も同様)

```
case （＜式＞）
 ＜ケース・アイテム＞
endcase
```

● ケース・アイテム(複数記述できる)

```
┌＜式＞ ：＜ステートメント＞
├＜式＞，＜式＞，… ：＜ステートメント＞
└default:＜ステートメント＞
```

《記述例》

```
case ( din ) // デコーダ
    0: dec = 4'b0001;
    1: dec = 4'b0010;
    2: dec = 4'b0100;
    3: dec = 4'b1000;
    default:
       dec = 4'bx;
endcase
```

● for 文

```
for （＜代入文＞；＜式＞；＜代入文＞）＜ステートメント＞
```

《記述例》

```
for( i=0; i<256; i=i+1 ) // メモリの初期化
    MEM[i] = 8'h0;
```

● while 文

```
while （＜式＞）＜ステートメント＞
```

《記述例》

```
i=0;
while( i<256 ) begin // メモリの初期化
    MEM[i] = 8'h0;
    i=i+1;
end
```

● repeat 文

```
repeat （＜式＞）＜ステートメント＞
```

注：repeat は固定回のループ．

《記述例》

```
i=0;
repeat( 256 ) begin // メモリの初期化
    MEM[i] = 8'h0;
    i=i+1;
end

repeat ( 8 ) @( posedge CK );
                      // 8クロック待ってTRIGを1に
TRIG = 1;
       // タイミング・コントロールと代入文を分離した例
```


● forever 文

```
forever ＜ステートメント＞
```

注：forever は無限ループ

《記述例》

```
initial begin //クロックの作成
    CK = 0;
    forever #(STEP/2) CK = ~CK;
end
```

● wait 文

```
wait(＜式＞）＜ステートメント＞
```

注：＜式＞が偽なら(0なら)待つ，真なら＜ステートメント＞を実行する．

● disable 文

```
┌disable ＜タスク名＞;
└disable ＜ブロック名＞;
```

注：指定したタスクやブロック(begin～end や fork～join)の処理を強制終了させる．

《記述例》

```
begin: LOOP
    for ( i=0; i<256; i=i+1 )
        if ( MEM[i]===8'hx )
            disable LOOP; // for ループからの脱出
```

```
  end
```

● force 文

```
force ＜代入文＞;
```

注：強制代入.

《記述例》

```
force BLK_A.CNT_EN = 1'b1; // 内部信号の強制イネーブル
```

● release 文

```
release ＜信号名＞;
```

注：強制代入の解除．一つの release 文で，信号は一つしか記述できない．

《記述例》

```
release blk_A.cnt_en; // 強制イネーブル解除
```

● タスク呼び出し(システム・タスクも同様)

表6　主要システム・タスク

画面表示	$display(P1, P2, ..., Pn); // 行末に改行あり $write(P1, P2, ..., Pn); // 行末に改行なし $monitor(P1, P2, ..., Pn); // 信号に変化あれば表示 $strobe(P1, P2, ..., Pn); // 全イベント終了後に表示
	上記はデフォルトで10進数表示．以下のようなデフォルトの基数を指定したタスクもある． $displayb (2進)，$displayo (8進)， $displayh (16進)
実行制御	$stop;　// 一時停止 $finish;　// 終了
関　数	$stime　// シミュレーション時刻 $time　// シミュレーション時刻 $random　// 乱数 $signed　// 符号付きに変換 $unsigned // 符号なしに変換
タイミング・チェック	$setup(...); // セットアップ時間 $hold(...);　// ホールド時間 $width(...); // 信号の幅
時間表示	$printtimescale(階層名.階層名.～.インスタンス名); // timescale の値を表示 $timeformat(単位番号,精度番号,文字列,最小表示けた); // %t のフォーマット指定
ファイルの一括読み出し/書き込み	$readmemh("ファイル名", memname, begin_addr, end_addr); $readmemb("ファイル名", memname, begin_addr, end_addr); $writememh("ファイル名", memname, begin_addr, end_addr); $writememb("ファイル名", memname, begin_addr, end_addr);

表7　出力フォーマット指定

%b	2進数	%e	E付き実数	%v	信号強度
%o	8進数	%f	10進実数	%l	ライブラリ情報
%d	10進数	%c	文字	%m	階層名
%h	16進数	%s	文字列	%t	時刻
画面表示システム・タスクなどでの出力フォーマット指定に使用する $display("文字列およびフォーマット", 引き数,引き数, ‥);					

```
├＜タスク名＞;
└＜タスク名＞ ( ＜式＞, ＜式＞, … );
```

表8　ファイル入出力システム・タスク

```
fd = $fopen("ファイル名", type);
$fclose(fd);
$fdisplay(fd, P1, P2, ..., Pn);
$fwrite(fd, P1, P2, ..., Pn);
$fmonitor(fd, P1, P2, ..., Pn);
$fstrobe(fd, P1, P2, ..., Pn);
c = $fgetc(fd);
code = $ungetc(c, fd);
code = $fgets(str, fd);
code = $fscanf(fd,  フォーマット文字列, 引き数);
code = $sscanf(str, フォーマット文字列, 引き数);
```

C言語のファイル入出力のライブラリ関数とほぼ同じものがある

● イベント起動

```
-> ＜イベント名＞
```

注：イベント信号の起動.

《記述例》

```
-> EVENT_SIG;
```

5 式

● 式

```
├<1次子>
├＜単項演算子＞ <1次子>
├＜式＞ <2項演算子> ＜式＞
├＜式＞ ? ＜式＞ : ＜式＞
└＜文字列＞
```

● 演算子

表9　演算子

算術演算	
+	加算，プラス符号
—	減算，マイナス符号
*	乗算
/	除算
%	剰余
**	累乗
ビット演算	
~	NOT
&	AND
|	OR
^	EX-OR
~^	EX-NOR
リダクション演算	
&	AND
~&	NAND
|	OR
~|	NOR
^	EX-OR
~^	EX-NOR

論理演算	
!	論理否定
&&	論理 AND
||	論理 OR
等号演算	
==	等しい
!=	等しくない
===	等しい(x，z も比較)
!==	等しくない(x，z も比較)
関係演算	
<	小
<=	小または等しい
>	大
>=	大または等しい
シフト演算	
<<	論理左シフト
>>	論理右シフト
<<<	算術左シフト
>>>	算術右シフト
そのほか	
?:	条件演算
{}	連接演算

● **演算の優先順位**

表10　演算優先順位

演算子	
! ~ + -	高
**	↑
* / %	
+ -	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ~&	
^ ~^	
| ~|	
&&	
||	↓
?:	低

表11　主要コンパイラ指示子

文字列置換	\`define <文字列1> <文字列2> \`define <文字列1>(引き数)<文字列2>(引き数)	●文字列1を文字列2に置換 ●参照時にも \` を付ける 例 \`define width 16 reg [\`width-1: 0] DBUS;
ファイル読み込み	\`include "ファイル名"	例 \`include "TASKS.v"
単位付け	\`timescale <時間> / <精度>	例 \`timescale 1ns/10ps
条件コンパイル	\`ifdef <文字列1> <記述1> \`elsif <文字列2> <記述2> \`else <記述3> \`endif	文字列1が定義済みなら記述1を，文字列2が定義済みなら記述2を，いずれも未定義なら記述3をコンパイルする 文字列の定義は，\`defineかシミュレーション起動時に行う
未宣言ネット対策	\`default_nettype none	ネット信号の未宣言使用を禁止 (モジュール間信号の宣言もれによる1ビット接続を抑制できる)

● **文字列**

" " で囲われた1行に収まる文字の集まり.

● **1次子**

```
├< 数値 >
├< 信号名 >
├< 信号名 > [< レンジ式 >]
├< 信号名 > [< 式 >] [< 式 >] ... [< 式 >]
├< 信号名 > [< 式 >] [< 式 >] ... [< 式 >] [< レンジ式 >]
├< 連接 >
├< ファンクション呼び出し >
└( < 式 > )
```

注：配列の [] は，宣言時に確保した次元の数だけ付加する.
ここの [] は省略可能を意味するのではなく，[] そのものを意味する.

● **数値**

```
├<10 進数値 >
├[< 符号なし数値 >] < 基数 > < 符号なし数値 >
└< 実数 >
```

● **基数**

次のいずれか(大文字でも可).

'b 'o 'd 'h 'sb 'so 'sd 'sh

● **ファンクション呼び出し**

```
├< ファンクション名 > ( < 式 >, < 式 >, … )
├< システム・タスク名 > ( < 式 >, < 式 >, … )
└< システム・タスク名 >
```

6 コンパイラ指示子

● バック・クォート(\`)で始まる.
● モジュールの内外に記述できる.
表11を参照.

7 コンフィグレーション

● **ライブラリ記述**

```
├library < ライブラリ名 > < ファイル・パス >,
│                            < ファイル・パス >, …
│            [ -incdir < ファイル・パス >,
│                          < ファイル・パス >, … ];
├include < ファイル・パス >;
└< コンフィグレーション宣言 >
```

● **コンフィグレーション宣言**

```
config < コンフィグレーション名;
  design [< ライブラリ名 >.] < セル名 >
                   [< ライブラリ名 >.] < セル名 > …
  < コンフィグレーション・ルール >
endconfig
```

● **コンフィグレーション・ルール**

```
├default liblist [< ライブラリ名 > < ライブラリ名 > …]
├instance < インスタンス名 > liblist
│                 [< ライブラリ名 > < ライブラリ名 > …]
├instance < インスタンス名 > use [< ライブラリ名 >.]
│                                    < セル名 > [:config]
├cell [< ライブラリ名 >.] < セル名 > liblist
│                 [< ライブラリ名 > < ライブラリ名 > …]
└cell [< ライブラリ名 >.] < セル名 > use
              [< ライブラリ名 >.] < セル名 > [:config]
```

注：コンフィグレーションは，モジュールの外部に記述する.

2